Response Time Analysis of a Middleware Event Demultiplexing Pattern for Network Services

Swapna S. Gokhale
Dept. of CSE
University of Connecticut
Storrs, CT 06269
Email: ssg@engr.uconn.edu

Aniruddha S. Gokhale
Dept. of EECS
Vanderbilt University
Nashville, TN
Email: a.gokhale@vanderbilt.edu

Jeff Gray
Dept. of CIS
U. of Alabama at Birmingham
Birmingham, AL 35294
Email: gray@cis.uab.edu

***Abstract*— Society is becoming increasingly reliant on the services provided by distributed, performance sensitive software systems. These systems demand multiple simultaneous quality of service (QoS) properties. A key enabler in recent successes in the development of such systems has been middleware, which comprises reusable building blocks. Typically, a large number of configuration options are available for each building block when composing a system end-to-end. The choice of the building blocks and their configuration options have an impact on the performance of the services provided by the systems. Currently, the effect of these choices can be determined only very late in the lifecycle, which can be detrimental to system development costs and schedules. In order to enable the right design choices, a systematic methodology to analyze the performance of these systems at design time is necessary. Such a methodology may consist of models to analyze the performance of individual building blocks comprising the middleware and the composition of these building blocks. As a first step towards building this methodology, this paper introduces a model of the Reactor pattern, which provides important synchronous demultiplexing and dispatching capabilities to network services and applications. The model is based on the Stochastic Reward Net (SRN) modeling paradigm. We illustrate how the model could be used to obtain the response time of a Virtual Private Network (VPN) service provided by a Virtual Router (VR).**

I. Introduction

Society is increasingly reliant on the services provided by distributed, performance-sensitive software systems. These systems demand multiple simultaneous quality of service (QoS) properties including predictability, controllability, and adaptability of operating characteristics for applications with respect to such features as time, throughput, accuracy, confidence, security and synchronization. A key enabler in recent successes in the development of such systems is *QoS-enabled middleware* [1]. Middleware comprises software layers that provide platform-independent execution semantics and reusable services that coordinate how application components are composed and interoperate. The flexibility and configurability offered by middleware is manifested in the large number of reusable software building blocks and configuration options, which can be used to compose and build large systems end-to-end. These building blocks embody good design practices called patterns [2], [3]. The choice of the patterns and their configuration options is driven by the context of the application. These choices have a profound impact on the performance of the provided service.

Current *ad hoc* techniques based on manually choosing the right set of building blocks and their configuration options are error-prone and may adversely impact performance, system costs and schedules, since most errors are caught very late in the lifecycle of the system development. It is desirable to have the ability to analyze the performance of individual building blocks and the composed system much earlier in the system lifecycle, thereby significantly lowering system testing costs as well as improving the correctness of the final developed system.

To address the challenge of system performance evaluation in the design phase, a systematic performance analysis methodology is necessary. This methodology would comprise developing performance models of the individual building blocks and their composition. The performance models are based upon well-known analytical/numerical modeling paradigms [4], [5], [6] and simulation techniques [7]. As a first step towards the development of such a methodology, this paper presents a model of the Reactor pattern [2], [3], which provides important synchronous demultiplexing and dispatching capabilities to network services and applications. The model is based on the Stochastic Reward Net (SRN) modeling paradigm [4]. We illustrate how the model can be used to obtain an estimate of the response time of a Virtual Private Network (VPN) service provided by a Virtual Router (VR) [8].

Paper organization: The paper is organized as follows: Section II presents the performance model of the Reactor pattern. Section III illustrates how the performance model of the Reactor pattern can be used to obtain the response time of a VPN service provided by a VR. Section IV offers concluding remarks and directions for future research.

II. Performance Model of the Reactor Pattern

In this section, we first provide an overview of the Reactor pattern followed by the SRN model of the Reactor pattern. The section also describes how the response time can be obtained from the SRN model.

A. Reactor Pattern in Middleware Implementations

Figure 1 depicts a typical event demultiplexing and dispatching mechanism documented in the Reactor pattern. The application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening for incoming events. On the occurrence of an event, the demultiplexer dispatches the event by making a callback to its associated application-supplied event handler. This is the idea behind the Reactor pattern, which provides synchronous event demultiplexing and dispatching capabilities.

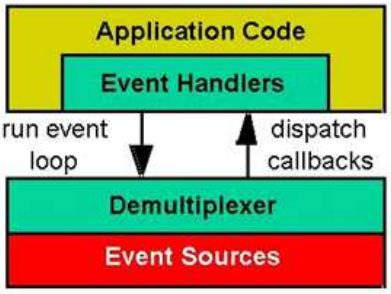

Fig. 1. **Event Demultiplexers in Middleware**

The Reactor pattern can be implemented in many different ways depending on the event demultiplexing capabilities provided by the underlying operating system and the concurrency requirements of the applications. For example, the demultiplexing capabilities of a Reactor could be based on the `select ()` or `poll ()` system calls provided by POSIX-compliant operating systems, or `WaitForMultipleObject ()` as found in the different flavors of Win32 operating systems. Moreover, the handling of the event in the event handler could be managed by the same thread of control that was listening for events leading to a single-threaded Reactor implementation. Alternatively, the event could be delegated to a pool of threads to handle the events leading to a thread-pool Reactor.

B. Characteristics of the Reactor Pattern

We consider a single-threaded, *select*-based implementation of the Reactor pattern with the following characteristics:

- The Reactor receives two types of input events with one event handler for each type of event registered with the Reactor.
- Each event type has a separate queue, which holds the incoming events of that type. The buffer capacity for the queue of type #1 events is denoted N_1 and of type #2 events is denoted N_2.
- Event arrivals for both types of events follow a Poisson process with rates λ_1 and λ_2, while the service times of the events are exponentially distributed with rates μ_1 and μ_2.
- In a snapshot, an event of type #1 is serviced with a higher priority over an event of type #2. In other words, when event handles corresponding to both event types are enabled in a snapshot, the event handle corresponding to type #1 is serviced with a priority that is higher than the event handle of type #2.

C. SRN Model

In this section we present the SRN model of the Reactor pattern. A Stochastic Reward Net (SRN) substantially extends the modeling power of Generalized Stochastic Petri Nets (GSPNs) [4], which are an extension of Petri nets [9]. A SRN is a modeling technique that is concise in its specification and closer to a designer's intuition about what a model should look like. SRNs have been extensively used for performance, reliability and performability analysis of a variety of systems [10], [11], [12], [13], [14], [15]. The work closest to the proposed research is reported by Ramani *et al.* [10], where SRNs are used for the performance analysis of the CORBA event service. A detailed overview of SRNs can be obtained from [4].

Figure 2 shows the SRN model for the Reactor pattern with the characteristics described in Section II-B. Table I summarizes the enabling/guard functions for the transitions in the net. The net on the left-hand side models the arrival, queuing and service of the two types of events. Transitions $A1$ and $A2$ represent the arrival of the events of type #1 and #2, respectively. Places $B1$ and $B2$ represent the queue for the two types of events. Transitions $Sn1$ and $Sn2$ are immediate transitions that are enabled when a snapshot is taken. Places $S1$ and $S2$ represent the enabled handles of the two types of events, whereas transitions $Sr1$ and $Sr2$ represent the execution of the enabled event handlers of the two types of events. An inhibitor arc from place $B1$ to transition $A1$ with multiplicity $N1$ prevents the firing of transition $A1$ when there are $N1$ tokens in place $B1$. The presence of $N1$ tokens in place $B1$ indicates that the buffer space to hold the incoming input events of the first type is full, and no additional incoming events can be accepted. The inhibitor arc from place $B2$ to transition $A2$ achieves the same purpose for type #2 events. The inhibitor arc from place $S1$ to transition $Sr2$ prevents the firing of transition $Sr2$ when there is a token in place $S1$. This models the prioritized service for an event of type #1 over event of type #2 in a given snapshot.

The net on the right of Figure 2 models the process of taking successive snapshots and prioritized service of the event handle corresponding to type #1 events in each snapshot. Transition $Sn1$ is enabled when there is a token in place $StSnpSht$, at least one token in place $B1$, and no tokens in place $S1$. Similarly, transition $Sn2$ is enabled when there is a token in place $StSnpSht$, at least one token in place $B2$, and no tokens in place $S2$. Transition $T\_SrvSnpSht$ is enabled when there is a token in either one of the places $S1$ and $S2$, and the firing of this transition deposits a token in place $SnpShtInProg$.

The presence of a token in the place $SnpShtInProg$ indicates that the event handles that were enabled in the current snapshot are being serviced. After these event handles complete execution, the current snapshot is complete and it is time

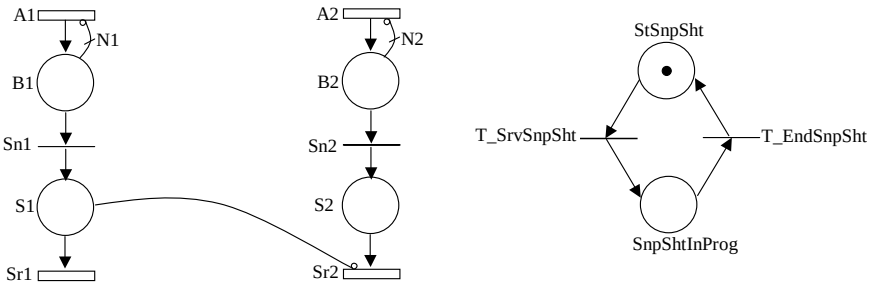

Fig. 2. **SRN model for the Reactor pattern**

to take another snapshot. This is accomplished by enabling the transition $T\_EndSnpSht$. Transition $T\_EndSnpSht$ is enabled when there are no tokens in both places $S1$ and $S2$. Firing of the transition $T\_EndSnpSht$ deposits a token in place $StSnpSht$, indicating that the service of the enabled handles in the present snapshot is complete, which marks the initiation of the next snapshot.

We now describe how the process of taking a single snapshot is modeled by the SRN model presented in Figure 2. We consider a scenario where there is one token in each one of the places $B1$ and $B2$, and there is a token in the place $StSnpSht$. Also, there are no tokens in places $S1$ and $S2$. In this scenario, transitions $Sn1$ and $Sn2$ are enabled. Both of these transitions are assigned the same priority, and any one of these transitions can fire first. Also, since these transitions are immediate, their firing occurs instantaneously. Without loss of generality, it can be assumed that transition $Sn1$ fires before $Sn2$, which deposits a token in place $S1$.

When a token is deposited in place $S1$, transition $T\_SrvSnpSht$ is enabled. In addition, transition $Sn2$ is already enabled. If transition $T\_SrvSnpSht$ were to fire before transition $Sn2$, it would disable transition $Sn2$, and prevent the handle corresponding to the second event type from being enabled. In order to prevent transition $T\_SrvSnpSht$ from firing before transition $Sn2$, transition $T\_SrvSnpSht$ is assigned a lower priority than transition $Sn2$. Because transitions $Sn1$ and $Sn2$ have the same priority, this also implies that the transition $T\_SrvSnpSht$ has a lower priority than transition $Sn1$. This ensures that in a given snapshot, event handles corresponding to each event type are enabled when there is at least one event in the queue.

After both event handles are enabled, transition $T\_SrvSnpSht$ fires and deposits a token in place $SnpShtInProg$. The presence of a token in the place $SnpShtInProg$ indicates that the event handles that were enabled in the current snapshot are being serviced. The event handle corresponding to type #1 event is serviced first, which causes transition $Sr1$ to fire and the removal of the token from place $S1$. Subsequently, transition $Sr2$ fires and the event handle corresponding to the event of type #2 is serviced. This causes the removal of the token from place $S2$. After both events are serviced and there are no tokens in places $S1$ and $S2$, transition $T\_EndSnpSht$ fires, which marks the end of the present snapshot and the beginning of the next one.

We obtain the response times of the events denoted R_1 and R_2 using the tagged customer approach [16]. In the tagged customer approach, an arriving event is tagged and its trajectory through the system is followed from entry to exit. The response time of the tagged event is then determined conditional to the state in which the system lies when the event arrives. The unconditional response time can be obtained as the weighted sum of the conditional response times, with the weights given by the steady state probabilities of being in each one of the states. Typically, the response time of an event consists of two pieces; namely, the time taken to service the event hereafter referred to as the "service time," and the time that the event must wait in the system before its service commences, hereafter referred to as "waiting time." In our case, the average service time of an incoming type #1 and type #2 event is given by $1/\mu_1$ and $1/\mu_2$, irrespective of the state in which the system lies when the event arrives. The waiting time, however, will depend on the system state. Next, we discuss how the conditional waiting time of each event type is determined.

The conditional waiting time for a tagged event of type #1 will depend on the state of the system, where the state is given by the number of tokens or markings of places $S1$, $S2$, $B1$ and $B2$. Of these four places, the markings of the places $S1$ and $S2$ determine the progress of the current snapshot, whereas, the markings of places $B1$ and $B2$ determine the state of the queue. The mean time taken to complete the current snapshot is given by the sum of two terms, the first term is the product of the number of tokens in place $S1$ and $1/\mu_1$, and the second term is the product of the number of tokens in place $S2$ and $1/\mu_2$. Even if there are no additional events in the queues, the current snapshot must be completed before the service of an incoming event of type #1 can begin. Hence, the time taken to complete the current snapshot contributes to the waiting time of the incoming or tagged type #1 event. In order to

TABLE I

ENABLING/GUARD FUNCTIONS

Transition	Guard function
$Sn1$	$((\#StSnpShot == 1)\&\&(\#B1 >= 1)\&\&(\#S1 == 0))?1 : 0$
$Sn2$	$((\#StSnpShot == 1)\&\&(\#B2 >= 1)\&\&(\#S2 == 0))?1 : 0$
$T\_SrvSnpSht$	$((\#S1 == 1)\|\|(\#S2 == 1))?1 : 0$
$T\_EndSnpSht$	$((\#S1 == 0)\&\&(\#S2 == 0))?1 : 0$

obtain the entire waiting time of a tagged type #1 event, the contribution of the queued events of type #1 and type #2 needs to be determined.

Let n_1 be the number of events of type #1 in the queue, and n_2 be the number of events of type #2 in the queue, when the tagged event of type #1 arrives. This implies that after n_1 snapshots the tagged event will be serviced. The following three possibilities arise between the relative values of n_1 and n_2. If $n_1 \leq n_2$, then only n_1 of the type #2 events need to be serviced before the service of the tagged type #1 event can commence, and hence the waiting time is given by $n_1(1/\mu_1 + 1/\mu_2)$. If $n_1 = n_2$, then n_1 events of type #1 and type #2 need to be serviced before the service of the incoming type #1 event can commence, and hence the waiting time is given by $n_1(1/\mu_1 + 1/\mu_2)$. If $n_1 > n_2$, then in the optimistic case, n_1 events of type #1 and n_2 events of type #2 need to be serviced before the service of the tagged event can commence. The optimistic case assumes that no additional events of type #2 arrive in the first n_1 snapshots. In the pessimistic case, however, $n_1 - n_2$ events of type #2 will arrive while the first n_2 events are being serviced. Thus, in the optimistic case, the waiting time will be $n_1/\mu_1 + n_2/\mu_2$, and in the pessimistic case, the waiting time will be $n_1(1/\mu_1 + 1/\mu_2)$. We consider the pessimistic case since that provides an upper bound on the response time. The pessimistic contribution of the queued events to the waiting time is given by the product of the number of tokens in place $B1$ and the sum of the reciprocals of μ_1 and μ_2. Thus, the overall response time of the tagged event will be given by the sum of two terms, the first term is $1/\mu_1$ times the sum of the tokens in places $S1$, $B1$ and 1, and the second term is given by the product of $1/\mu_2$ and the sum of the number of tokens in place $S2$ and $B1$. The contribution of the queued events to the waiting time of the tagged event of type #2 can also be determined using similar reasoning, with an additional consideration given to the prioritized service provided to event of type #1 over an event of type #2 in each snapshot. The reward rates to obtain the response time of the events of type #1 and type #2 are summarized in Table II.

In the model of the reactor pattern described above, the arrival, service and failure distributions are assumed to be exponential. For certain types of applications, this assumption may not hold. For example, for safety-critical applications, events may occur at regular intervals, in which case the arrival process is deterministic. In addition to the deterministic distribution, the arrival, service and failure processes may also follow any other non-exponential or general distribution. There are two ways to consider non-exponential distributions in the SRN model. In the first method, a non-exponential distribution can be approximated using a phase-type approximation [4], and the resulting SRN model can then be solved using SPNP [17]. In the second method, the model can be simulated using discrete-event simulation incorporated in SPNP.

III. CASE STUDY: VPN SERVICE USING VIRTUAL ROUTER

In this section we describe how the SRN model of the Reactor pattern presented in Section II-C can be used to estimate the response time of a Virtual Private Network (VPN) service provided by a Virtual Router (VR).

Figure 3 illustrates the architecture of a provider-provisioned virtual private network (PPVPN) [18] using a VR. A VR is a software/hardware component that is part of a physical router called the provider edge (PE) router. A VR contains the mechanisms to provide highly scalable, differentiated levels of services in VPN architectures. Multiple VRs can reside on a PE device. VRs can be arranged in a hierarchical fashion within a single PE as shown in Figure 3. Moreover, an entity acting as a service provider for an end customer might itself be a customer of a larger service provider. VRs may also use different backbones to improve reliability or to provide differentiated levels of service to customers.

Customer edge (CE) devices wishing to join a VPN connect to a VR on the PE device. A VR can multiplex several distinct CEs belonging to the same VPN session. A VR may use tunneling mechanisms to use multiple routing protocols and link layer protocols, such as IPSec, GRE, and IP-in-IP, to connect with the CEs. A totally different set of protocols and tunneling mechanisms could be used for inter-VR or VR-backbone communication. These tunneling mechanisms can also be the basis for differentiated levels of service as well as to provide improved reliability. A VR also comprises firewall capabilities.

We consider a scenario where a VR is used to provide VPN services to two organizations, with each organization having a customer edge (CE) router connected to the VR. The employees of each organization issue VPN set up and tear down services to the VR via CEs. Also, the VR offers differentiated levels of service, with organization #1 receiving prioritized service over organization #2. It is important that these requests be serviced in a reasonable amount of time. Additionally, it is also critical to obtain an estimate of what the response time might be at the time the VPN service is provisioned.

In order to implement the VPN service, a Reactor pattern with the characteristics described in Section II-B can be used to (de)multiplex the events. The SRN model of the Reactor

TABLE II

REWARD ASSIGNMENTS FOR RESPONSE TIME

Event type	Reward rate
#1	return($\#B1 < N1?1/\mu_1 * (\#S1 + \#B1 + 1) + 1/\mu_2 * (\#S2 + \#B1) : 0$)
#2	return($\#B2 < N2?1/\mu_2 * (\#S2 + \#B2 + 1) + 1/\mu_1 * (\#S1 + \#B2 + 1) : 0$)

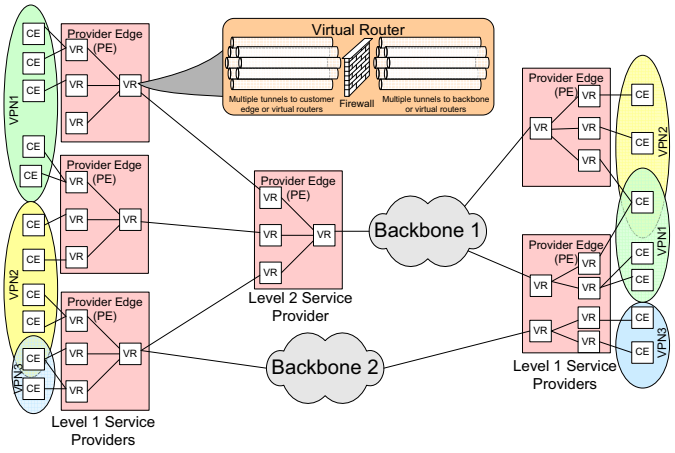

Fig. 3. **VPN Architecture using Virtual Routers**

pattern can be used to obtain an estimate of the response time of the requests. Towards this end, we designate the requests originating from organization #1 as events of type #1 and requests originating from organization #2 as events of type #2. We set the buffer capacities for both types of events to five, and the service rates of both types of events to 2.0/sec. The arrival rate of both types of events were set to 0.4/sec. The expected response times for type #1 and type #2 events obtained by solving the SRN model using SPNP [17] are 0.83 seconds and 1.33 seconds, respectively. It can be observed that the response time for set up and tear down requests for organization #2 is higher than the response time for requests from organization #1 due to the prioritized service provided to organization #1 in each snapshot.

In this case study, estimates of the expected response times were obtained for fixed settings of the parameters. At design time, however, it is rarely the case that the exact values of the parameters are known. As a result, in the design phase it becomes necessary to analyze the sensitivity of the estimates to the values of the parameters. Sensitivity analysis can also be used to establish bounds on the performance estimates and for the provisioning of resources. We now demonstrate how the SRN model could be used for sensitivity analysis with relative ease. For the sake of illustration, we analyze the sensitivity of the response time estimates to the arrival rates of the events. Towards this end, we vary the arrival rates of the events from 0.4/sec to 2.0/sec one at a time, and obtain the expected response time estimates for each value of the arrival rate. Figures 4 and 5 show the expected response times as a function of event arrival rates.

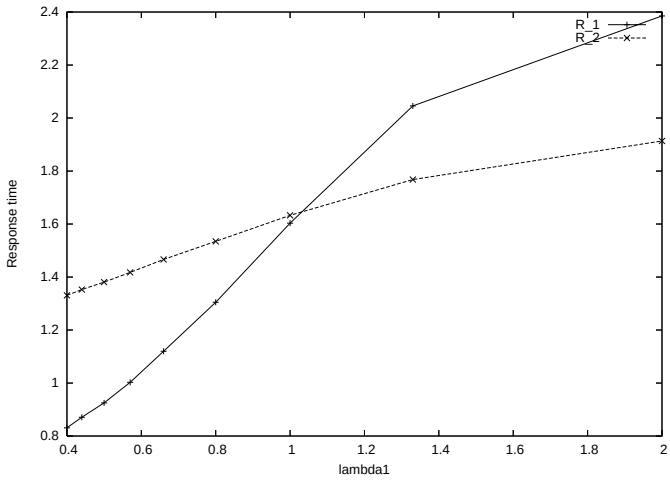

Fig. 4. **Response Time as a function of λ_1**

The plot in Figure 4 shows the expected response time as a function of λ_1, and the plot in Figure 5 shows the expected response time as a function of λ_2. Figure 4 indicates that as λ_1 increases, the expected response times for both types of events increase. At approximately $\lambda_1 = 1.0$, the expected response times for both types of events is close. However, as λ_1 increases beyond 1.0/sec the expected response time of type #1 events is higher than the expected response time of type #2 events. Thus, in effect, requests from organization #2 are receiving better service than requests from organization #1. On the other hand, Figure 5 indicates that the expected response time of both types of events increases as λ_2 increases, for the

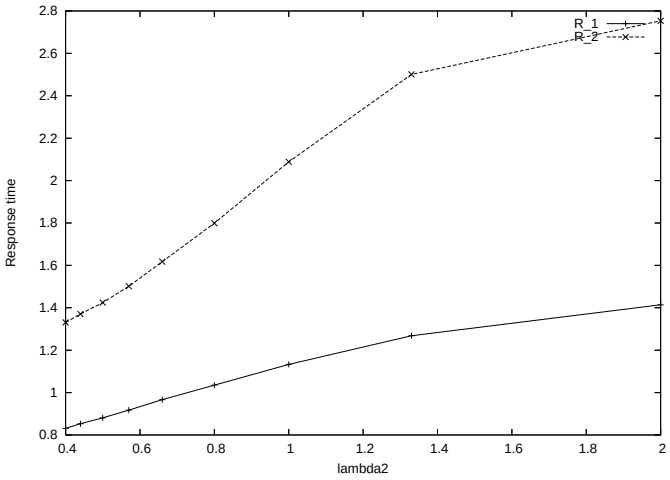

Fig. 5. **Response Time as a function of λ_2**

entire range of λ_2. In this case requests from organization #1 continue to receive better service than requests from organization #2, although the absolute value of the expected response time increases as λ_2 increases.

IV. Conclusions and future research

In this paper we presented a performance model of the Reactor pattern which offers the important synchronous demultiplexing and dispatching capabilities in middleware. The model was based on the Stochastic Reward Net (SRN) modeling paradigm. We illustrated how the performance model could be used to obtain an estimate of the response time of a VPN service provided by a Virtual Router (VR). Our future research consists of empirically validating the response time estimates obtained from the performance model. Developing and validating the performance models of other middleware building blocks and the composition of these building blocks is also a topic of future research.

Acknowledgments

This research was supported by the following grants from the National Science Foundation (NSF): Univ. of Connecticut (CNS-0406376 and CNS-SMA-0509271), Vanderbilt Univ. (CNS-SMA-0509296) and Univ. of Alabama at Birmingham (CNS-SMA-0509342).

References

[1] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering*, J. Marciniak and G. Telecki, Eds. New York: Wiley & Sons, 2001.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[3] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[4] A. Puliafito, M. Telek, and K. S. Trivedi, "The evolution of stochastic Petri nets," in *Proc. of World Congress on Systems Simulation*, Singapore, September 1997, pp. 3–15.

[5] H. Choi, V. Kulkarni, and K. S. Trivedi, "Markov Regenerative Stochastic Petri Net," *Performance Evaluation*, vol. 20, no. 1–3, pp. 337–357, 1994.

[6] G. Horton, V. Kulkarni, D. Nicol, and K. S. Trivedi, "Fluid stochastic Petri nets: Theory, application and solution techniques," *Journal of Operations Research*, vol. 405, 1998.

[7] The VINT Project, "Network Simulator - NS-2," www.isi.edu/nsnam/ns, 1996.

[8] P. Knight, H. Ould-Brahim, and B. Gleeson, "Network based VPN Architecture using Virtual Routers," *IETF Network Working Group Internet Draft, draft-ietf-l3vpn-vpn-vr-02.txt*, pp. 1–21, Apr. 2004.

[9] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[10] S. Ramani, K. S. Trivedi, and B. Dasarathy, "Performance analysis of the CORBA event service using stochastic reward nets," in *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems*, October 2000, pp. 238–247.

[11] O. Ibe, A. Sathaye, R. Howe, and K. S. Trivedi, "Stochastic Petri net modeling of VAXCluster availability," in *Proc. of Third International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, 1989, pp. 142–151.

[12] O. Ibe and K. S. Trivedi, "Stochastic Petri net models of polling systems," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 9, pp. 1649–1657, December 1990.

[13] H. Sun, X. Zang, and K. S. Trivedi, "A stochastic reward net model for performance analysis of prioritized DQDB MAN," *Computer Communications, Elsevier Science*, vol. 22, no. 9, pp. 858–870, June 1999.

[14] O. Ibe and K. S. Trivedi, "Stochastic Petri net analysis of finite–population queueing systems," *Queueing Systems: Theory and Applications*, vol. 8, no. 2, pp. 111–128, 1991.

[15] J. Muppala, G. Ciardo, and K. S. Trivedi, "Stochastic reward nets for reliability prediction," *Communications in Reliability, Maintainability and Serviceability: An International Journal Published by SAE Internationa*, vol. 1, no. 2, pp. 9–20, July 1994.

[16] B. Melamed and M. Yadin, "Randomization procedures in the computation of cumulative-timed distributions over discrete-state markov process," *Operations Research*, vol. 32, no. 4, pp. 926–944, July-August 1984.

[17] C. Hirel, B. Tuffin, and K. S. Trivedi, "SPNP: Stochastic Petri Nets. Version 6.0," *Lecture Notes in Computer Science 1786*, 2000.

[18] A. Nagarajan, "Generic Requirements for Provider Provisioned Virtual Private Network (PPVPN)," in *IETF Network Working Group Request for Comments, RFC 3809*, A. Nagarajan, Ed. IETF, June 2004, pp. 1–25.